EVENT CORRELATION IN SPECTRUM® AND OTHER COMMERCIAL PRODUCTS

September 2000

APRISMA™
MANAGE WHAT MATTERS

Introduction

Event correlation over the enterprise has become an issue of paramount importance in enterprise management. Event correlation can be defined loosely as the process of making sense of a very large number of events, where "making sense" entails throwing some of them away, observing cause-and-effect relations between others, inferring an alarm from a set of related events, or identifying the culprit event in a misbehaving enterprise. See Figure 1.

The event correlation task is somewhat analogous to the interaction between the brain and the five senses. The senses deliver a very large number of impressions to the brain. However, our brain has evolved such that we ignore the great majority of these signals. Further, our brain has the capacity to interpret our impressions. For example, we may infer the simple idea of "impending trouble" based on our surroundings.

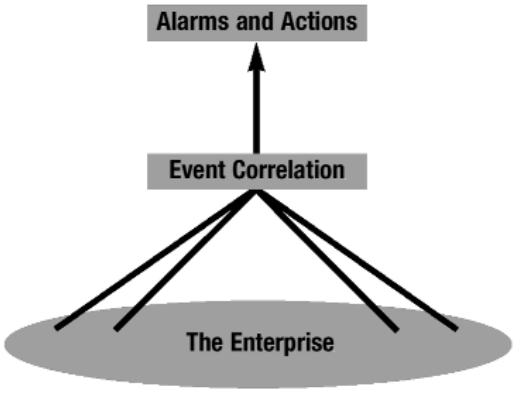


Figure 1. Event Correlation over the Enterprise

In this paper we examine five approaches to the event correlation task:

Rule-based reasoning (BMC Patrol, Tivoli TME, and others)

Model-based reasoning (SPECTRUM)

State transition graphs (NerveCenter)

Codebooks (InCharge)

Case-based reasoning (SpectroRx)

Rule-based Reasoning

A common approach to the event correlation task is to represent knowledge and expertise in a rule-based reasoning (RBR) system (a.k.a. expert system, production system, or blackboard system). See Figure 2. An RBR system consists of three basic parts:

- A working memory
- A rule base
- A reasoning algorithm

The working memory consists of facts. The rule base represents knowledge about what other facts to infer, or what actions to take, given the particular facts in working memory. The reasoning algorithm is the mechanism that actually makes the inference.

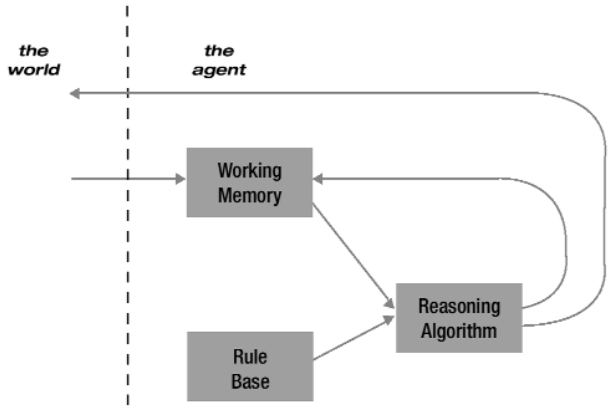


Figure 2. The Basic Structure of an RBR

The best way (albeit very simple) to think about the operation of the reasoning algorithm is to recall the classic Modus Ponens inference rule in elementary logic:

A	**A fact in working memory**
If A then B	**A rule in the rule base**
Therefore, B	**An inference made by the reasoning algorithm**

In this simple example, since the antecedent A of the rule "If A then B" matches fact A in the working memory, we say that the rule fires and the directive B is executed.

Note that B can be several kinds of directive:

- Add a new fact to working memory.
- Perform a test on some part of the enterprise and add the result to working memory.
- Query a database and add the result to working memory.
- Query an agent and add the result to working memory.
- Execute a control command on some enterprise component (e.g., reconfigure a router, or prohibit a certain class of traffic over a link or network).
- Issue an alarm via some alarm notification medium.

For example:

Rule 1: *if* load(N1, t1) = high *and*
packet_loss(N1, t1) = high *and*
connection_failure(C3, S, t1) = true *and*
connection_failure(C4, S, t1) = true

then add_to_memory(problem(t1) = traffic congestion)

Rule 2: *if* problem(t1) = traffic congestion

then add_to_memory(Show traffic by category from t1–10 to t1)

Regardless of the particular directive, after the reasoning algorithm makes a first pass over the working memory and the rule base, the working memory becomes enlarged with new facts. The enlargement might be a result of the directives, or it might be a result of the monitoring agents entering new events in working memory over time. In either case, on the second pass there might be other rules that fire and offer new directives and therefore new facts, and so on for each subsequent pass.

At this juncture we should be able to appreciate the sort of complexity entailed by representing knowledge with RBR systems. The good news is that rules are rather intuitive, as is the basic operation of an RBR system. But the bad news is that it is critical to come up with a correct set of rules that behave in the way that we conceptualize them. This problem shows up especially when subsequent passes over the memory and rule base issue unplanned directives, or seem to be going nowhere, or when the control algorithm gets caught in a non-ending loop.

The general consensus in the industry regarding the use of RBR systems is this:

> If the domain that the RBR system covers is small, non-changing, and well understood, then it's a good idea to use the RBR method. If one tries the RBR approach when these three conditions do not hold, one is asking for trouble.

Therefore, using an RBR system to develop an event correlator that covers the entire domain of the enterprise is not a good idea.The enterprise is large, dynamic, and generally hard to understand.

But this doesn't mean that RBR systems don't have a place in enterprise management. For example, let us think about a single computer system as opposed to an entire enterprise. A single computer system is a much smaller entity than an enterprise. It is reasonable, then, to consider an RBR system to perform event correlation over this small domain.

In fact this is precisely the way that most vendors who build computer monitoring agents do it—for example, BMC Patrol,Tivoli TME, Computer Associates Unicenter, and Platinum ProVision. Most of these systems are one-iteration type systems. The reasoning algorithm periodically makes a pass over the memory and the rule base and checks to see if any event (or some set of events) should be escalated to an alarm. Such events include repetitious failures of log-on attempts and thresholds for parameters such as disk space and CPU usage.

Model-based Reasoning

Let us consider a simple example to explain how model-based reasoning (MBR) works and how it differs from rule-based reasoning. The author often uses this example in lectures and professional talks, and it usually gets the idea across and stimulates further discussion. Refer to Figure 3 as we work through the example.

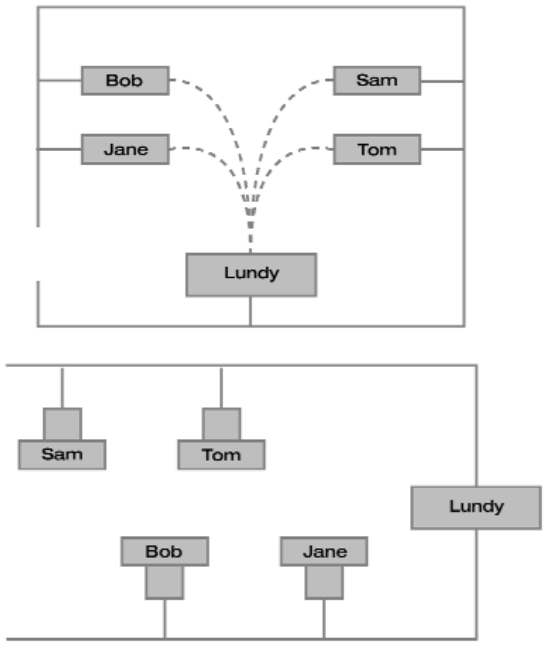


Figure 3. A Classroom with Students and a Lecturer

Suppose we have a classroom with several students, and Lundy is the lecturer. Imagine that each student in the class is a model (i.e., a mirror image in software) of some real computer system outside the door. Jane and Tom are models of NT servers, Bob and Sam are models of UNIX workstations, and so forth.

Now imagine that the Jane,Tom, Bob, and Sam models have a way to communicate with their real-world counterparts.They ping their counterparts every ten minutes to make sure they are alive, and also collect information about their general health.

The classroom is a model of a subnet. Lundy is a model of the router that hosts the subnet, and Lundy is in communication with the router. Thus there is a real-world system outside the door, and the classroom is a representation of the system.

Lundy and the students are happily pinging their real-world counterparts when all of a sudden Jane pings her NT server and doesn't get a response. After two more pings with no response, Jane sends a message to Lundy asking if he has heard from his router. If Lundy answers yes, then Jane infers that there is a fault with her NT server and raises an alarm accordingly. But if Lundy answers no, then Jane reasons that probably her NT server is in good shape, and the real fault is with the router.

The thrust of the example is that event correlation is a collaborative effort among virtual intelligent models, where the models are software representations of real entities in the enterprise.

A formal description of a MBR system for event correlation is as follows:

- An MBR system represents each component in the enterprise as a model.
- A model is either a representation of a physical entity (e.g., a hub, router, switch, port, computer system), or a logical entity (e.g., LAN, MAN, WAN, domain, service, business process).
- A model that represents a physical entity is in direct communication with the entity (e.g., via SNMP).
- A description of a model includes three categories of information: attributes, relations to other models, and behaviors.
    - Examples of attributes for device models are ip address, MAC address, alarm status, and myriad others.
    - Examples of relations among device models are connected to, depends upon, is a kind of, and is a part of.
    - An example of a behavior is: *If I am a server model and I get no response from my real-world counterpart after three tries, then I request status from the model to which I am connected and then make a determination about the value of my alarm status attribute.*
- Event correlation is a result of collaboration among models (i.e., a result of the collective behaviors of all models).

The best example in the industry of the MBR approach is SPECTRUM, which contains model types (a.k.a. classes in object-oriented terminology) for roughly a thousand kinds of physical and logical entities. These model types are used as patterns for creating models of the actual entities in an enterprise. Each model type contains generic attributes, relations, and behaviors that occurrences of the type would exhibit.

The first thing one does after installing SPECTRUM is to run SPECTRUM's Auto-discovery. Auto-discovery discovers and models the entities in the enterprise, and then fills in the generic characteristics of each model with actual data. Then, as monitoring happens in real time, the models collaborate with respect to their predefined behaviors in order to realize the event correlation task.

What happens if there exists an entity for which a model type is not available? There are two ways to approach this situation. First, one can exploit the *is a kind of* relation among models. In object-oriented terminology, this relation is called inheritance.

Suppose we have a generic model of a router replete with placeholders for attributes, relations, and behaviors that most all routers share. Then we can define a derivative model of the router and say that it is a kind of generic router. The derivative model inherits the characteristics of its parent, and the job is done.

The second way is to implement a new model type in C++ code and link it with the existing model type hierarchy. Note that this method requires an experienced software engineer.

An alternative way to implement event correlation is to use a SPECTRUM product called SpectroWatch. SpectroWatch is an RBR system as described in the previous section. It is integrated with SPECTRUM and it augments SPECTRUM's core MBR method. One can use SpectroWatch to formulate rules that describe how events are mapped into alarms.

The advantage of this alternate approach is that it is easy to do, since there is a GUI that guides one through the process and C++ programming is not required.The disadvantage is that it may suffer the usual deficiencies of RBR systems: If the domain is large, then the performance of event correlation can be jeopardized.

State Transition Graphs

The key concepts in the STG approach are a token, a state, an arc, a movement of a token from one state to another state via an arc, and an action that is triggered when a token enters a state.

To see how the STG apparatus works, consider the scenario in the preceding section. There was a subnet hosted by a router, and the subnet contained several UNIX workstations and NT servers. One of the NT servers (Jane's server) failed to respond to a ping, but the actual problem was that the router (Lundy's router) had failed.

We want the event correlator to be able to reason that Jane's NT server is only apparently at fault, and Lundy's router is the real fault. Figure 4 shows an STG that can reason this through.

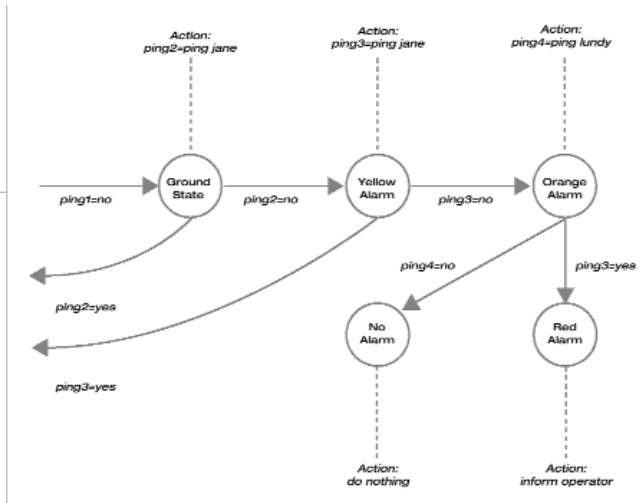


Figure 4. A Sample State Transition Graph

The first failure of Jane's NT server causes a token to move into a Ground state. An action of the Ground state is to ping Jane's' NT server after one minute elapses. If the ping doesn't show that Jane's NT server is alive, then the token either moves to the Yellow Alarm state, or the token falls off the board.An action of Yellow Alarm is to ping Jane's NT server a second time. If the action returns no, then the token moves to the Orange Alarm state, where the action is to ping Lundy's router. Depending upon the state of Lundy's router, the token moves to either the No Alarm state or the Red Alarm state, and the appropriate action is taken accordingly.

Note that the STG in Figure 4 covers a single domain of interest— Jane's NT server. However, an enterprise might contain thousands of diverse kinds of components, and thus it might seem that a large number of STGs may be required to cover the enterprise. Fortunately, since a generic STG can apply to components of the same type, there is no need to build separate STGs for like components.

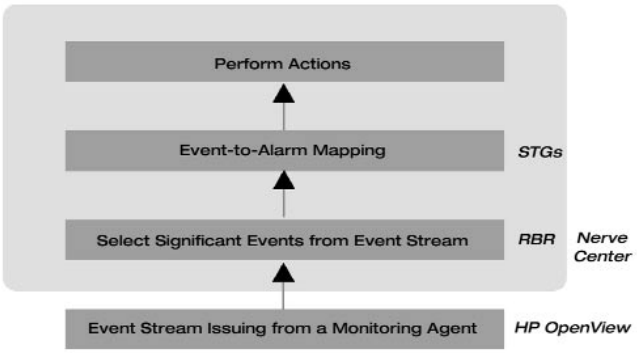


Figure 5. The Structure of the OpenView/NerveCenter Integration

Further, an action of one state may be to confer with the output of a separate STG. For example, the action of the Orange Alarm state in Figure 4 was to ping Lundy's router. Instead, it could have been to ping a STG that covers the router. Note that this design comes close to the collaboration of virtual models described earlier in the MBR approach.

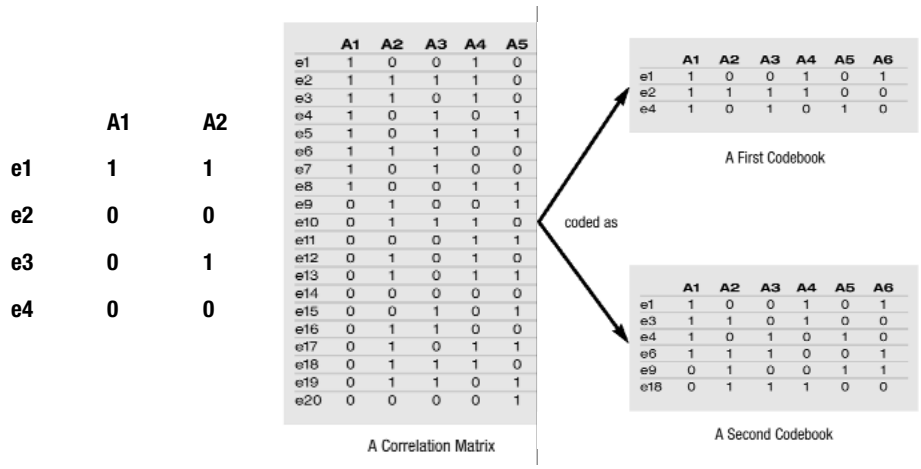
The best example of the STG approach is SeaGate's NerveCenter. NerveCenter is typically integrated with HP OpenView, although it can operate in standalone mode by communicating directly with managed devices via SNMP. See Figure 5.

The first thing we should notice in Figure 5 is that NerveCenter uses the classic RBR method to select significant events from the OvenView event stream, and only these events are passed to a set of STGs to perform the event-to-alarm mapping function. In commercial literature, one often sees NerveCenter described as a rule-based system, which is somewhat misleading. NerveCenter uses two kinds of representations: rules and STGs.

Codebooks

The major concepts in the codebook approach to event correlation are a correlation matrix, coding, a codebook, and decoding. To see how the codebook apparatus works, let us consider a simple example.

Consider a small domain of interest in which there are four kinds of events: e1, e2, e3, and e4; and two kinds of alarms: A1 and A2. Now, suppose we know the sets of events that cause each alarm. We organize this information in a correlation matrix as follows:



	A1	A2
e1	1	1
e2	0	0
e3	0	1
e4	0	0

	A1	A2	A3	A4	A5
e1	1	0	0	1	0
e2	1	1	1	1	0
e3	1	1	0	1	0
e4	1	0	1	0	1
e5	1	0	1	1	1
e6	1	1	1	0	0
e7	1	0	1	0	0
e8	1	0	0	1	1
e9	0	1	0	0	1
e10	0	1	1	1	0
e11	0	0	0	1	1
e12	0	1	0	1	0
e13	0	1	0	1	1
e14	0	0	0	0	0
e15	0	0	1	0	1
e16	0	1	1	0	0
e17	0	1	0	1	1
e18	0	1	1	1	0
e19	0	1	1	0	1
e20	0	0	0	0	1

A Correlation Matrix

coded as

	A1	A2	A3	A4	A5	A6
e1	1	0	0	1	0	1
e2	1	1	1	1	0	0
e4	1	0	1	0	1	0

A First Codebook

	A1	A2	A3	A4	A5	A6
e1	1	0	0	1	0	1
e3	1	1	0	1	0	0
e4	1	0	1	0	1	0
e6	1	1	1	0	0	1
e9	0	1	0	0	1	1
e18	0	1	1	1	0	0

A Second Codebook

Figure 6. A Correlation Matrix and Two Derivative Codebooks

This matrix tells us that an occurrence of event e1 indicates A1, whereas a joint occurrence of e1 and e3 indicates A2.

"Coding" transforms the matrix into a compressed matrix, which is called a codebook. It is not hard to see that the matrix above can be compressed into a much simpler codebook:

	A1	A2
e1	1	1
e3	0	1

The codebook tells us that only e1 and e3 are required to determine whether we have A1, A2, or neither. Now, with this compressed codebook, the system is ready to perform event-to-alarm mapping. This is called decoding. The system simply reads events off an event stream and compares them with the codebook in order to infer alarms.

Note that we could also perform the event-to-alarm mapping function with the original correlation matrix as well as the compressed codebook. However, it is not hard to appreciate the gain in speed when we use the latter. The codebook tells us to be on the lookout for two events instead of four events, and there is considerably less codebook lookup as compared with the original matrix.

Second, the codebook approach allows one to compress a very large correlation matrix, which is hard for the human mind to comprehend, into codebooks that are more comprehensible. To see this, consider the correlation matrix and the two derivative codebooks in Figure 6.

The first codebook collapses the correlation matrix into three events: e1, e2, and e4.The codebook can distinguish among all six alarms; however, in some cases it can guarantee distinction only by a single event. For example, A2 and A3 are distinguished by e4. A lost or spurious generation of e4 will result in a potential decoding error (i.e., incorrect event-to-alarm mapping).

The second codebook resolves this problem by considering six events: e1, e3, e4, e6, e9, and e18.The second codebook is such that a lost or spurious generation of any two events can be detected and any single-event error can be corrected.

There are a few things one has to pay attention to when using the codebook approach to event correlation:

First, how does one come up with the original correlation matrix? If the original matrix is correct, then we have an opportunity to compress the matrix into a codebook that hides noise and irrelevant events. However, if it isn't correct, we run into the familiar problem of garbage-in garbage-out.

Assume that the original matrix is indeed correct. Can we be assured that a high-performance codebook will be generated? Consider the following simple example:

	A1	A2
e1	0	1
e3	1	0

A codebook for this matrix will be equal to the matrix. In general, the compression factor of a correlation matrix depends upon the patterns of data collected in it. Depending upon the patterns, the compression factor may be anywhere from very high to very low.

Finally, how do we choose from multiple codebooks? In general, given any correlation matrix of reasonable size, there will be a very large number of possible codebooks. For example, our discussion of Figure 6 showed that the simplest codebook may not be resilient against errors. On the other hand, a codebook that is resilient to errors may sacrifice simplicity and understandability.

The best example in the industry of the codebook approach is InCharge, developed by System Management Arts (SMARTS). InCharge is typically integrated with either HP OpenView or IBM NetView.

InCharge includes an event modeling language based on classic object-oriented techniques, including class/subclass development, inheritance, and event definition. For example, a class TCPPort and a class UDPPort may inherit the general attributes of a class Port. However, the event PacketLossHigh for a TCPPort has a different definition than PacketLossHigh for a UDPPort.

InCharge contains a generic library of networking classes. Given these classes, one may derive domain-specific classes by adding the appropriate attribute and instrumentation statements to produce an accurate model of the domain. Finally, one adds event definitions to the model. Note that this approach comes close to the MBR approach described earlier.

Thus far we have considered four approaches in the industry for performing event correlation over the enterprise: RBR systems, MBR systems, state transition graphs, and codebooks. Each method represents knowledge and reasoning in different ways. One thing that they have in common is that they rate rather low with respect to learning and adaptability. Our final method, case-based reasoning, compares favorably in this regard.

Case-based Reasoning

The basic idea of CBR is to recall, adapt, and execute episodes of former problem solving in an attempt to deal with a current problem. Former episodes of problem solving are represented as cases in a case library. When confronted with a new problem, a CBR system retrieves a similar case and tries to adapt the case in an attempt to solve the outstanding problem. The experience with the proposed solution is embedded in the library for future reference. The general CBR architecture is shown in Figure 7.

A challenge of the CBR approach is to develop a similarity metric so that useful cases can be retrieved from a case library. We would not want the system to retrieve the case that simply has the largest number of matches with the fields in the outstanding case. Some of the fields in a case are likely to be irrelevant, and thus misguide the system.Thus, relevance criteria are needed to indicate what kinds of information to consider given a particular problem at hand.

An example of a relevance criterion is the following:

The solution to the problem "response time is slow" is relevant to bandwidth, network load, packet collision rate, and paging space.

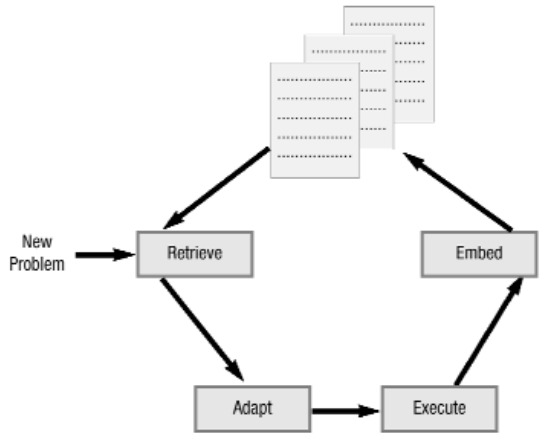


Figure 7. The General CBR Architecture

Note that relevance criteria are not the same as rules in a RBR system. Relevance rules simply tell the system what cases to look at, but not what to do with them.

How does one acquire relevance criteria? Current research involves the application of machine learning algorithms (such as neural networks and induction-based algorithms) to an existing case library. The output of the algorithm is a list of relevance rules. For the present, however, a pragmatic approach is to handcraft relevance rules and test them manually.

An additional challenge for the CBR approach is to develop adaptation techniques by which the system can tweak an old solution to fit a new problem, although the new problem might not be exactly like the old problem. We discuss several kinds of adaptation techniques below.

Consider an outstanding problem,"response time is unacceptable," and imagine that only one source case is retrieved from the case library, as shown below. In this example, the resolution is page_space_increase = A, where A is a solution variable that holds the amount by which to increase the page space of a server, determined by the function f.

Problem: response time = F
Solution: A = f(F), page_space_increase = A
Solution status: good

"Parameterized adaptation" is a method for adjusting the solution variable of an outstanding problem relative to the problem variable, based on the relation between the solution and problem variables in a source case. Everything else being equal, an outstanding problem "response time = F*" should propose the solution page_space_increase = A*, where F* and A* stand in the same relation as F and A in the source case. The proposed solution in the outstanding case, therefore, would be like the following:

Problem: response time = F*
Solution: A* = f(F*), page_space_increase = A*
Solution status: ?

How does one acquire functions like f? The safest method is to handcraft and test them, and in general there are several ways to represent f. The simplest is a look-up table, where values of A* not in the table are calculated by interpolation. Also, learning f from a collection of historical performance data can be looked upon as a function approximation task, and thus lends itself to neural network methods, which are generally good at function approximation, e.g., counter-propagation and back-propagation.

Note also that f does not have to be a function per se. For other kinds of problems, f might be a sequence of steps or a decision tree. Suppose a retrieved case holds a simple procedure as follows:

Solution: reboot(device = client1)

where reboot is a process and client is the value of the variable device. Suppose this case is just like an outstanding case except that in the outstanding case the value of device is server1. Thus, the advised solution is:

Solution: reboot(device = server1).

This adaptation method is called “adaptation by substitution.

” Now, since it is impolite to reboot a server without warning, one might wish to prefix the step “issue warning to clients” to the advised solution and enter the case back into the case library. In this example, the proposed solution is adapted manually by a user, so this technique is called “critic-based adaptation.”

There exist several generic CBR systems in the industry—e.g., CBR Express from the Inference Corporation and SpectroRx from Aprisma. SpectroRx is an add-on application for SPECTRUM. As described earlier, SPECTRUM performs the event correlation task using the MBR method. Once a fault is identified, however, there remains the problem of finding a repair for the fault. Clearly, past experiences with similar faults are important, and this is just the kind of knowledge that SpectroRx allows one to develop.

An interesting story regarding SpectroRx is as follows. Version I of SpectroRx was shipped in 1995 with an empty case base. In the industry, this is sometimes called a knowledge shell.The user was expected to build an initial seed case library manually, after which the system would expand and become increasingly fine-tuned with use.

The problem was that users weren’t too keen on the idea of first having to build a seed case library, although they liked the general idea of CBR. Many requests came in to ship SpectroRx with a generic seed case library. But now the engineers who developed SpectroRx had a problem. How could one build a generic case library, when any two enterprises are likely to be quite different—each having different components, services, and configurations?

After much research to try to understand what a generic case library would look like, and to determine whether there should be just one or several generic case libraries, an engineer proposed a simple, ingenious solution: Suppose we take the event-to-alarm mapping knowledge in SPECTRUM and transform it into a set of cases?

Part of the beauty of this solution is that although SPECTRUM contains more than a thousand models for popular enterprise components, it is never the case that an enterprise will be composed of each kind of component.Typically, one to two hundred models are actually instantiated. Therefore, the seed case library is based on the event-to-alarm mapping knowledge that is related to only these instantiated models.

Version II of SpectroRx was shipped in 1996 with a case library that holds just one case, which says:

Problem: Your case library is empty.

Solution: Depress the Execute Solution button and we will build a seed case library for you in about 10 minutes.

Solution Status: ?

This solution solves the problem elegantly. For any two enterprises, the seed case library will be especially tailored to reflect the components, services, and configuration that are unique for each enterprise.

Summary and Conclusion

In this paper we looked at five paradigms by which to build an event correlator for the enterprise: rule-based reasoning, model-based reasoning, state transition graphs, codebooks, and case-based reasoning. We saw that RBR is not appropriate for an enterprise-wide event correlator, but it well suited for smaller domains such as computer monitoring and control. MBR, STG, and codebooks are used in the products SPECTRUM, NerveCenter, and InCharge respectively. MBR is built into SPECTRUM. NerveCenter and InCharge are standalone applications that are often integrated with HP OpenView or IBM NetView. Finally, we saw neither method is capable of learning and adaptation.The last method, case-based reasoning, shows promise in this regard.

Further Reading

Frey and Lewis. Multi-level Reasoning for Managing Distributed Enterprises and their Networks. In *Integrated Network Management V.* Chapman and Hall. 1997.

Hasan, Sugla, and Viswanathan. A Conceptual Framework for Network Management Event Correlation and Filtering Systems. In *Integrated Network Management VI.* IEEE Press. 1999.

Houk,Calo, and Finkel. Towards a Practical Alarm Correlation System. In *Integrated Network Management IV.* Chapman and Hall. 1995.

Jakobson and Weissman. Real-Time Telecommunication Management: Extending Event Correlation with Temporal Constraints. In *Integrated Network Management IV.* Chapman and Hall. 1995.

Katker and Paterok. Fault Isolation and Event Correlation for Integrated Fault Management. In *Integrated Network Management V.* Chapman and Hall. 1997.

Kliger, Yemini,Yemini, Ohsie, and Stolfo. A Coding Approach to Event Correlation. In *Integrated Network Management IV.* Chapman and Hall. 1995.

Lewis.A Case-Based Reasoning Approach to the Management of Faults in Communications Networks. In *Proceedings of IEEE INFOCOM'93.* IEEE Press. 1993.

Lewis. *Managing Computer Networks: A Case-Based Reasoning Approach.* Artech House 1995.

Liu, Mok, and Yang. Composite Events for Network Event Correlation. In *Integrated Network Management VI.* IEEE Press. 1999.

Mayer, Kliger, Ohsie, and Yemini. Event Modeling with the MODEL Language. In *Integrated Network Management V.* Chapman and Hall. 1997.

Nygate. Event Correlation Using Rule and Object Based Techniques. In *Integrated Network Management IV.* Chapman and Hall. 1995.

APRISMA™
MANAGE WHAT MATTERS

121 Technology Drive | Durham, NH 03824 | www.aprisma.com
Phone: (603) 334-2100 | Sales: (877) 437-0291 | Fax: (603) 334-2784

APR-WP10-1200